the Availability Digest

www.availabilitydigest.com

Choosing a Business Continuity Solution
Part 2 – Data Replication
August 2011

In Part 1 of this series, we reviewed various concepts of availability. We pointed out that systems can be highly available, exhibiting minutes of downtime per outage, or continuously available with seconds of downtime per outage. If a backup data center exists to continue operations following a disaster of some sort, the backup data center can provide disaster recovery or disaster tolerance. With disaster recovery, IT services can be restored, though it may take days or weeks. With disaster tolerance, IT services continue uninterrupted following a disaster.

Fundamental to all highly available and continuously available architectures is data replication. Such availability requires redundancy. It is data replication that maintains redundant databases in synchronization so that an up-to-date database copy is immediately available following an outage. Part 2 of this series[1] explores the various replication technologies and their strengths and weaknesses.

Data Replication - The Fundamental Force Behind Availability

Improving your availability via data replication depends upon having at least two nodes, each being capable of hosting a database. Typically, each node also can host the application that is being protected.[2]

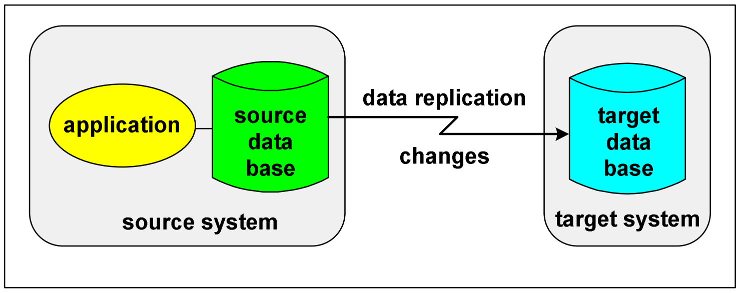

Figure 1: Data Replication

[1] This series of articles is a reprint of a Gravic, Inc., white paper and is published with the permission of Gravic.
[2] There are replication architectures in which the target node is a "data bunker" whose purpose is to safe-store data. In the event of a primary-system failure, the data bunker is used to bring the database of the backup system up-to-date. Depending upon the architecture used, these approaches can offer little to no data loss. However, they typically have very long recovery times (many hours to days). In this paper, we will focus on replication architectures that provide low to no data loss but that have very fast recovery times.

As shown in Figure 1, the purpose of data replication is to keep a target database in synchronism with a source database that is being updated by a source application.

We talk about the *source database* hosted by the *source node* and the *target database* hosted by the *target node*. The two nodes comprise the distributed data-processing system. As an application makes changes (inserts, updates, and deletes) to its local database (the source database), those changes are sent immediately over a communication channel to the target system, where they are applied to the target database (Figure 2). The target database typically resides on another independent node that may be hundreds or thousands of miles away. We call the facility that gathers changes made to the source database and applies them to the remote target database a *replication engine*.

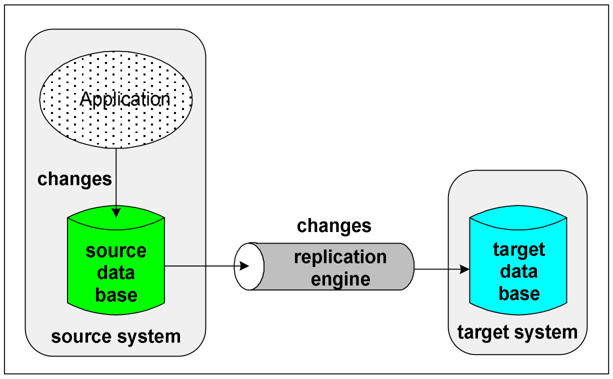

Figure 2: Data-Replication Engine

Data-Replication Techniques – An Overview

Data-replication engines can be categorized in several ways:[3]

- <u>Hardware versus Software Replication</u> – With *hardware replication*, the replication engine is implemented via low-level device drivers, typically in the storage subsystem. A *software-based replication engine* can reside either in the storage subsystem or, as is usually the case in the systems we will discuss, in the processing nodes themselves.

- <u>Asynchronous versus Synchronous Replication</u> – *Asynchronous replication* sends source database changes to the target database without impacting the source application. Changes are sent after-the-fact from a queue of changes maintained on the source node. The application and the data-replication engine are decoupled from each other via the change queue.

  Synchronous replication allows no changes to be made unless they can be made to all database copies simultaneously. The application and the data-replication facilities are coupled to each other. As we shall see, depending upon the approach used, they may be either tightly coupled or loosely coupled.

- <u>Unidirectional versus Bidirectional Replication</u> – With *unidirectional replication*, changes are sent in just one direction from a source database to a target database. With *bidirectional replication*, both databases can be active; and changes made to either are

[3] *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.
Breaking the Availability Barrier II: Achieving Century Uptimes with Active/Active Systems, AuthorHouse; 2007.
Breaking the Availability Barrier III: Active/Active Systems in Practice, AuthorHouse; 2007.

replicated to the other. In this case, each database is both a source and a target.[4] In either case, replication can be either asynchronous or synchronous.

Hardware Versus Software Replication

Hardware Replication

Replicate on Cache Flush

Hardware replication is usually implemented in the storage-system controller. It replicates disk blocks as they are written to the source disk, thus guaranteeing that the contents of the target disk are always identical to the source disk.

However, disk blocks are typically only written to disk when they are flushed from the disk's cache. There is no logical order to the disk-write sequence since other factors control cache flushing - disk blocks that are the least recently used are flushed to disk when cache space is needed for new blocks that must be read from disk. As a consequence, the target disk is not guaranteed to be consistent. Target disk blocks may be partially split. Indices may exist without the rows or records to which they refer. Children may exist without parents. The data is consistent in cache, but the target-disk image is generally useless. As a result, applications cannot use the target database for any application processing. Should the source node fail, a lengthy recovery process is required to bring the target database into a useful, consistent state.

Additionally, because of the cache-flushing issue, large amounts of data may be lost due to a source-system failure even if synchronous replication is used, as any data still in cache will not have been flushed at the time of failure.

Replicate on Cache Update

Some storage controllers replicate changes as they are made to cache regardless of whether they have been physically written to the source disk or not. The replication of cache updates can ensure the logical consistency of the target database since changes are replicated to the target system as soon as they are made at the source system. Therefore, the target database can provide a consistent view of the source database.

Hardware replication, whether based on disk flushing or cache updating, typically sends blocks of changes to the target. In some cases, the controller can compress data to only those bytes that have changed. Hardware replication is limited to specific hardware and may not be possible in your configuration.

The two replication techniques described above generally require the identical storage technology down to the version to be used at both the source and the target. They also do not typically allow the target database to be opened by applications at the same time that replication is taking place, thus preventing their use in active/active systems. Therefore, hardware replication is not an option if recovery times measured in seconds or minutes is to be achieved. As a consequence, hardware replication will be discussed no further in this paper.

[4] Bidirectional replication may cause problems with certain applications that are not designed to be distributed. In this case, an alternative architecture that can be used is a "sizzling-hot standby." Bidirectional replication is configured with applications running on both systems. However, all transactions are sent only to one system, while the other system acts as a hot standby. Should the primary system fail, all that needs to be done is to reroute transaction activity to the standby system. Since it is already configured for replication, the failed system can be easily restored to operation once it is repaired.

Software Replication

In the high-availability systems that we will consider, higher-level software carries out the replication task. A data-replication engine running on the source and target systems performs the replication. Only in this way can the continuously available active/active systems that we will describe later be implemented.

Software-replication engines typically read changes from a change queue of some sort and send them to the target system to update its database. So long as updates are made to the target system in the same order as they were made at the source system, the target database will be consistent and usable by other applications. Some high-performance replication engines are multithreaded to improve replication throughput. In these engines, resynchronizing facilities reorder updates that may be received out-of-order from the various threads before the updates are applied to the target database.

Software replication may be by event, by transaction, or by request. *Event replication* replicates DML (data manipulation language) events as they occur. DML events include insert, update, and delete operations. Event replication in some cases may also replicate DDL (data definition language) operations that affect the database's data structure and schema.

Transaction replication replicates entire transactions, either one operation at a time as they occur or as a group of operations once the transaction has committed on the source. When replayed at the target, the transaction is either committed or, if the entire transaction is not received, is aborted.

Request replication replicates the entire request, which is reprocessed in its entirety by the target system.

Asynchronous versus Synchronous Replication

Asynchronous Replication

An *asynchronous data-replication engine* is completely transparent to the applications running in the source node. As shown in Figure 3, it extracts changes made to the source database from a change queue and sends them after-the-fact to the target database. The replication engine makes changes to the target-database copy somewhat later than they were made to the source database. The result is that the databases are synchronized, but the target database copy lags the source database by a short interval. This interval is known as the *replication latency* of the data-replication engine.

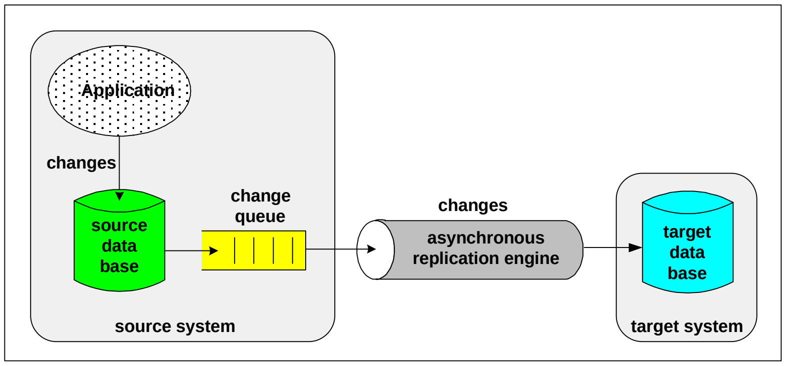

Figure 3: Asynchronous Replication Engine

The replication latency associated with an asynchronous replication engine creates an issue that must be considered when using such technology. This issue is data loss following a failure of the source node. Any changes that were in the replication pipeline and that did not make it to the target node may be lost. The data loss in this case is the replication latency of the replication engine, which is typically measured in the tens or hundreds of milliseconds.

Synchronous Replication

A *synchronous data-replication engine* solves the asynchronous-replication problem of data loss following a node failure. Synchronous replication makes no permanent changes to any database copy unless those changes can be applied to all database copies. Therefore, should a node or the network fail, no data is lost. Synchronous replication can satisfy RPOs (Recovery Point Objectives) of zero (that is, no data loss).

Synchronous replication has its own issue, and that is *application latency.* Since the application must wait for the transaction's data to be safe-stored and optionally applied to all database copies in the application network, the source application's transaction completion is delayed.

There are two primary synchronous-replication methods that we will describe – *dual writes* and *coordinated commits*.

When using *dual writes*, all copies of the database are included within the scope of the transaction (Figure 4). The application must wait for each database update to complete across the network before proceeding to the next update. It must then wait for the transaction to be committed across the network before informing the application that the transaction is complete. This delay is a function primarily of the communication latency between the nodes (which is related to the distance separating the nodes) and of the size of the transaction. Thus, the nodes typically must be near each other – such as in the same campus or metropolitan area – and connected by very fast media such as fiber, which may not allow the degree of separation required for proper disaster tolerance.

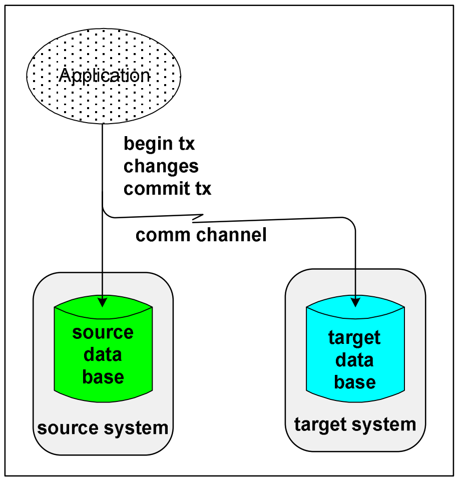

Figure 4: Dual Writes

The *coordinated-commit* method for synchronous replication minimizes application latency. A coordinated-commit replication engine is a combination of synchronous-replication and asynchronous-replication techniques. The coordinated-commit replication engine registers as a voting member of the source system's transaction. As shown in Figure 5, changes are sent to the target database asynchronously so that they do not impact the application. It is only at commit time that the coordinated-commit replication engine must wait for one replication latency to check with the target database to ensure that it can vote "yes" on the transaction.

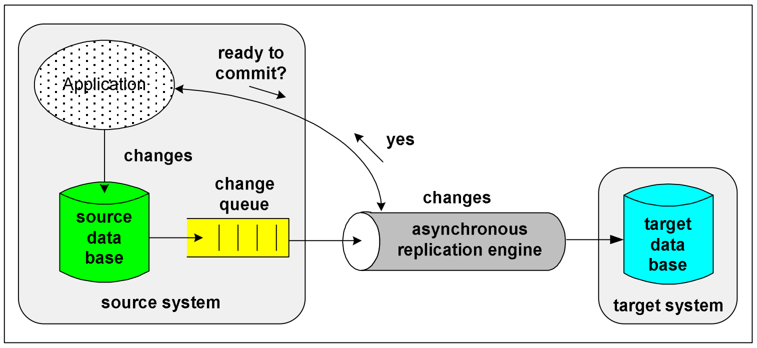

Figure 5: Coordinated Commits

Thus, the coordinated-commit technique imposes an application latency of one replication latency plus one channel latency as it coordinates with its target database at commit time. Even if the nodes are separated by thousands of miles, application latency with this method can be as small as tens of milliseconds.

Unidirectional versus Bidirectional Replication

Unidirectional Replication and Active/Passive Systems

A *unidirectional data-replication engine* replicates data in one direction – from a source database to a target database. Figures 3, 4, and 5 are examples of unidirectional asynchronous- and unidirectional synchronous data replication engines. Unidirectional replication is often used to maintain a passive backup system in synchronism with an active production system. These systems are known as *active/passive systems*.

Unidirectional replication for active/passive systems will always have much longer recovery times than bidirectional-replication active/active systems, described next.

Bidirectional Replication and Active/Active Systems

A *bidirectional data-replication engine* replicates data in *both* directions between two databases. Each database is acting both as a source database and as a target database. Any change made to one database is reflected in the other database via data replication. Therefore, every node in the application network has a current copy of the application database and can participate in the application. Since each processing node is actively engaged in the application, such systems are known as *active/active systems*.

Advantages of Active/Active Systems

Active/active systems provide a wide range of advantages compared to active passive/systems. Advantages include:

- There are fewer users affected by a failure. In an active/passive architecture, all users are down. In an active/active system, only the users connected to the failed node are affected.

- Failover of affected users can be very rapid, supporting RTOs (Recovery Time Objectives) measured in subseconds to seconds. This is because all that needs to be done is to reroute transactions or to reconnect users to a surviving node following a node failure.

- Failover can be periodically and safely tested since all nodes are known to be operational because they are actively processing transactions. When a failover does occur, it is to a known working system, providing peace of mind for management.

- Planned downtime can be eliminated by taking down one node at a time, performing upgrade or maintenance activities on it, and then returning it to service.

- An application can use all available processing capacity. There is no idle standby system.

- Capacity can be added simply by adding nodes to the application network. There is no need to replace existing systems with larger systems. Alternatively, an existing node can be replaced with a larger or smaller node to increase or decrease the capacity of the system. The new node’s database is synchronized with the application database, and users are then rerouted to it. At this point, the old node can be taken out of service.

- Load easily can be balanced by rerouting some transaction activity to underutilized nodes.

- Processing nodes can be located near clusters of users, thereby providing data locality and a reduction in response time.

- A node can be located in a “lights-out” facility since its failure will not deny users access to the application’s services.

- If an application cannot run in a distributed environment, it can still be beneficial to run it in an active/active environment. However, all activity is routed to only one node, which provides all transaction processing. This configuration is called a “*sizzling-hot standby*.” It resolves the application distributed-processing issues but retains all of the continuous-availability features of an active/active system.

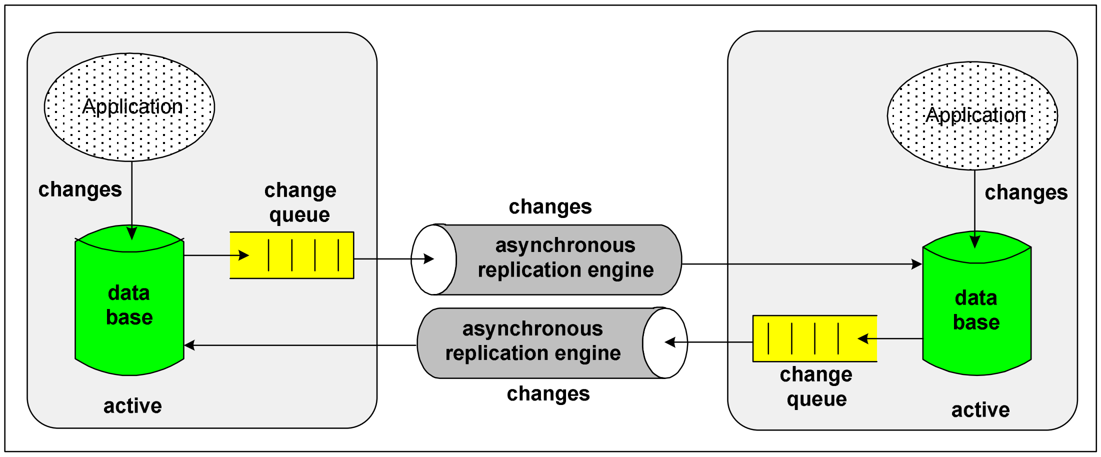

Figure 6: Bidirectional Replication Engine

Figure 6 shows an example of an asynchronous bidirectional data-replication engine. In effect, two unidirectional asynchronous replication engines (see Figure 3) are each replicating in opposite directions. However, the two replication engines are not independent; and they are more complex than a unidirectional replication engine. For one thing, each side must cooperate to ensure that a change received via replication is not replicated back to the source of the change, a condition known as *data oscillation* or *ping-ponging*.[5]

In addition, with asynchronous replication, it is possible that the same data item might be changed in each copy of the application database within the replication-latency interval. If this should happen, neither database will know of the conflict. Each will replicate its change to the other database, and the replicated changes will overwrite the original change in each database. Now both databases are different, and both are wrong. This is called a *data collision*.

Data Collisions

There are some applications in which data collisions can be ignored. For instance, perhaps a temporary database divergence is not important; and the databases will be resynchronized when that data item is once again updated.

Some applications avoid data collisions. For instance, an application that is insert-only will not suffer data collisions.

[5] Replicated changes are updates to the database and are reflected in the target system’s change queue. Since changes are replicated from the change queue, unless some protection is provided, replicated changes will be replicated back to the source system; and the process will be repeated. See Strickler, G., et al., “*Bi-directional Database Replication Scheme for Controlling Ping-Ponging*,” United States Patent 6,122,630; Sept. 19, 2000.

Some active/active architectures avoid collisions. One frequently used method is to partition the database between the nodes. For instance, if the database can be partitioned by customer range so that only one node updates any given partition of customer data, data collisions cannot occur. In this case, the application must route all updates to the proper node that owns the customer data partition being updated.

The preferable method is for data collisions to be *avoided*. However, if data collisions are possible, they must be detected and resolved. There are several techniques by which the replication engine can automatically detect and resolve data collisions. They include:[6]

- Detection – Detection is generally accomplished by sending with the source change a row version of some sort that identifies the version of the row that the source system is changing. The row version can take many forms, such as a timestamp, a version number, or a before image of the row. If the target system finds that the row version being updated by the source system is not the same as the current row version in the target database, a data collision has occurred.

- Resolution – Once the replication engine has detected a data collision, it must make a decision as to which change to accept and which to reject. The decision rule must provide consistency so that all nodes will make the same decision. Otherwise, the database copies may diverge. Many collision resolution rules exist. The selection of a resolution algorithm depends upon the application and how it processes data.

  Examples of collision-resolution rules include:

  - Choosing the update that carries the latest (or earliest) timestamp.

  - Choosing the update that was made by the node with the highest precedence.

  - Using relative replication, in which operations such as add and subtract are replicated rather than row contents. Since these operations are commutative (that is, they can be executed in any order and arrive at the same result), data collisions do not result in database divergence.

  - Choosing the update according to specialized business rules bound into the data-replication engine.

  - In those cases where automatic resolution is not possible, collisions will have to be resolved manually.

Bidirectional synchronous replication avoids data collisions since the replication engine must acquire locks on all copies of the data item across the network before it can change (or as it changes) any of them. Therefore, only one application at a time can change a data item, thus avoiding collisions. A bidirectional synchronous replication engine can be implemented using two unidirectional, coordinated-commit synchronous replication engines (see Figure 5), one for each direction, as shown in Figure 7.

[6] Chapter 3, Asynchronous Replication, and Chapter 4, Synchronous Replication, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, AuthorHouse; 2004.

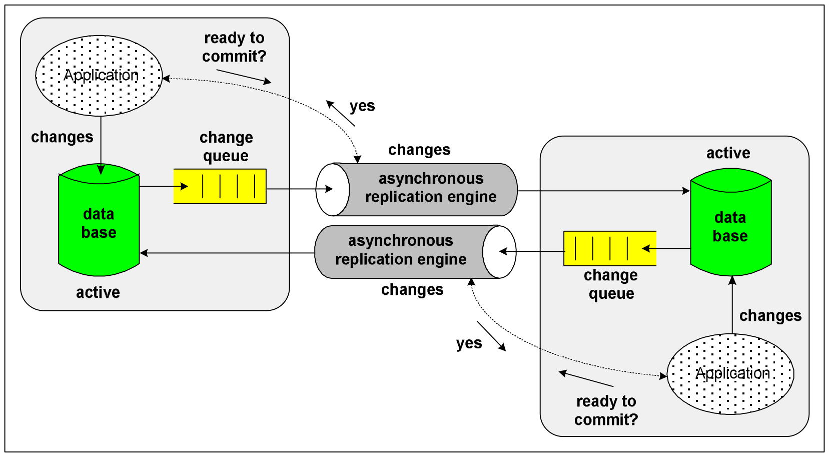

Figure 7: Bidirectional Synchronous Replication Engine

Summary

A wide variety of data-replication technologies are in use today. They support active/passive systems, in which the backup system is passively standing by ready to take over if it is needed. Also supported are active/active systems, in which all systems are actively involved in the application. It can take minutes to hours to fail over to a passive backup in an active/passive system. Failover in active/active systems is immediate since all that needs to be done is to reroute transaction traffic from the failed system to surviving systems.

In Part 3, we look more carefully at the various architectures used to achieve a wide range of availabilities. We compare these technologies to each other in a Business Continuity Continuum.